Programmeerimise põhikursus

ITI0010

Loengu sisu

Objektid ja klassid: sissejuhatus ja ülevaade

Milleks OO
Objektid ja klassid
Inheritance
Näited
Overloading
Abstract classes

Programming in the large II: OOP

OOP: object-oriented programming.

Essentially, **OOP is a set of special techniques to simplify programming** (mostly for large systems: it will NOT help programming very small systems). **Especially good for graphical user interfaces**!

Main techniques in OOP:

Information hiding: other programmers cannot call some of my functions or change some of my global variables.

Combining data and functions in a class: functions may access data in their instance of a class only.

Function overloading: same function name can be called with different parameters (needs different implementations as well)

Hierarchies of classes and inheritance: lower classes contain (inherit) everything from higher classes (and can modify some).

OOP main concepts

OOP: object-oriented programming. **Classes as software modules,** ie easy-to-use, ready-made programming blocks.

The central concept of object-oriented programming is the **object**, which is a kind of module containing data and subroutines.

DATA + FUNCTIONS (SUBROUTINES)

The point-of-view in OOP is that an **object is a kind of self-sufficient entity** that has an internal state (the data it contains) and that can respond to messages (calls to its subroutines).

A mailing list object, for example, **has a state consisting of a list of names and addresses**.

If you send it a message (call a function) **telling it to add a name**, it will respond by modifying its state to reflect the change.

If you send it a message (call a function) **telling it to print itself**, it will respond by printing out its list of names and addresses**.**

OOP ideologically

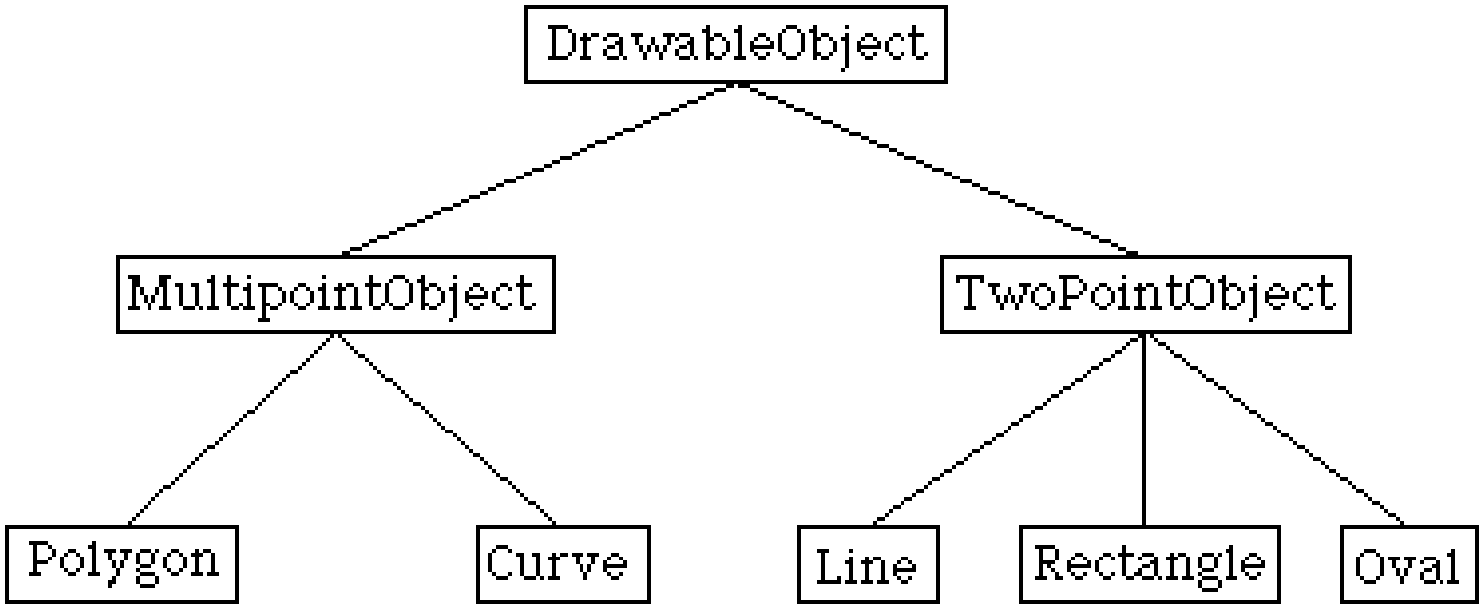
OOP: “Programming consists of designing a set of objects that somehow model the problem at hand. Software objects in the program can represent real or abstract entities in the problem domain.“

This is supposed to make the design of the program more natural and hence easier to get right and easier to understand

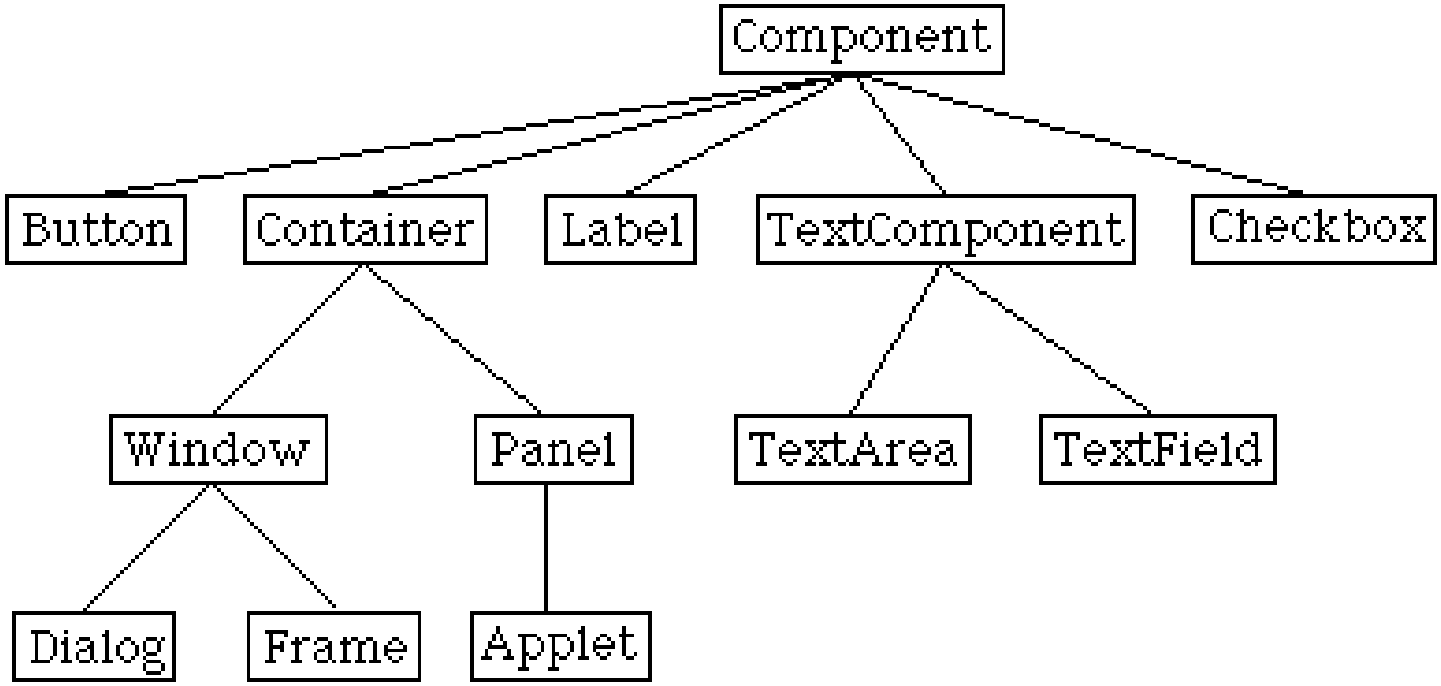
NB! Main bulk of programming time and ingenuity always goes into PROGRAMMING IN THE SMALL: while, if, assignments etc.

However, using OOP for putting together a program makes the design, structure, readability, modifiability etc better and easier.

Objects: families and inheritance



GUI objects: inheritance



OOP: classes

Every object belongs to some class. **We say that the object is an instance of that class**.

In OOP, classes are **templates** for making objects. That is, the class of an object specifies what sort of data that object contains and what behaviors it has.

The **data** of an object is contained in a set of variables called the instance variables, and the **behaviors** of the object exist as a set of subroutines called the instance methods of the object.

Static and non-static:

In Java in particular, it is the **non-static** variables and methods in the class that determine the contents of an object, while the class's static variables and methods are members of the class itself rather than of the objects that belong to that class.

Static vars and functions are not really OO.

OOP: types and actual data

It is important to understand that the class of an object determines the **types** of the instance variables; however, the actual data is contained inside the individual object, not the class. Thus, each object has its own set of data.

For example, there might be **a class named Student:**

Suppose that this class has a non-static variable called name, of type String.

When this class is first loaded by the system, there are no actual names -- just a kind of idea that students have names.

Every time a Student object is created using the Student class as a template, the object will include an instance variable called name of type String.

Since each Student includes its own name variable, each student can have a different name. Similarly, if objects of class Student have instance variables to represent test grades, then each Student object has its own set of grades.

An **instance method** belonging to an object has direct access to that particular object's instance variables.

OOP: student class: a template

```
public class Student {
    public String name;  // Student's name
    public int ID;       // unique ID number
    public double test1, test2, test3; // grade

    public double getAverage() { // comp avg
        return (test1 + test2 + test3) / 3;
    }

    private static int nextUniqueID = 0;

    public static int getUniqueID() {// give id
              nextUniqueID++;
              return thisID;
    }

}  // end of class Student
```

OOP: create objects I: types

How to **define vars** for objects from classes:

A class name is considered to be a **type**, similar to built-in types such as `int` and `boolean`.

So, if you have a class, such as `Student`, you can use the class name to declare variables of that class:

```
Student std;  // declare variable
              // std of type Student
```

However, declaring a variable does not create an object!

In Java, no variable can ever hold an object.
A variable can only hold a reference to an object.

You should think of objects as floating around independently in the computer's memory.

Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a **reference** or **pointer** to the object. In effect, a reference to an object is the address of the memory location where the object is stored.

OOP: create objects II: actual objects

Objects are actually created by an operator called **`new`**, which creates an object and returns a reference to that object.

For example, assuming that `std` is a variable of type `Student`, the assignment statement

```
std = new Student();
```

would create a new object of type `Student` and store a reference to that object in the variable `std`.

The instance variables and methods of the new object could then be accessed through `std`, as in "**`std.test1`**".

OOP: null reference

It is possible for a variable like `std`, whose type is given by a class, to refer to no object at all. We say in this case that `std` holds a **null reference**. The null reference can be written in Java as "**`null`**".

You could assign a null reference to the variable `std` by saying

```
std = null;
```

and you could test whether the value of `std` is null by testing

```
if (std == null) . . .
```

If the value of a variable is null, then it is illegal to refer to instance variables or instance methods through that variable.

For example, if the value of the variable `std` is `null`, then it would be illegal to refer to `std.test1`.

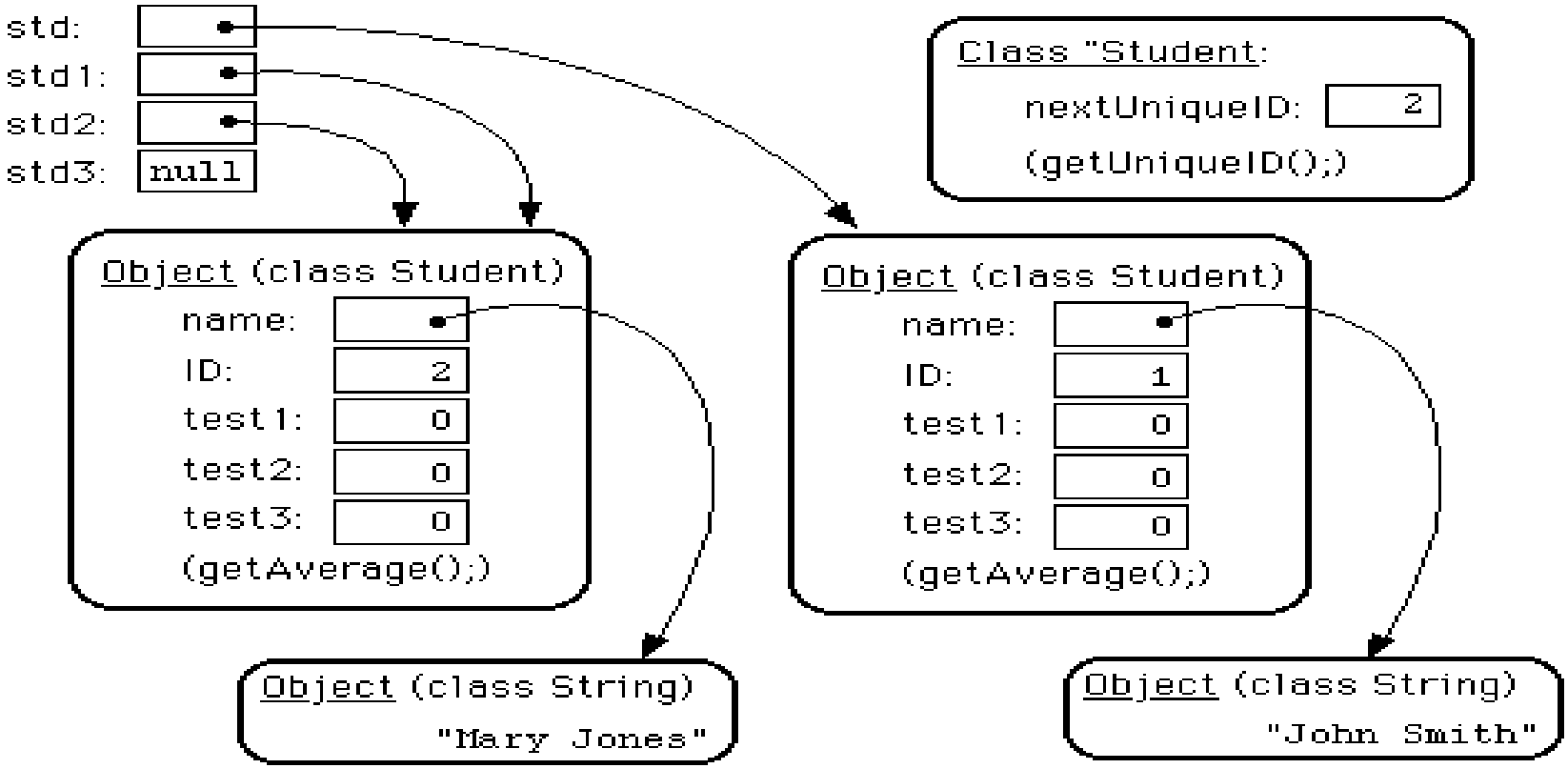
If your program attempts to use a null reference illegally like this, the result is an error called a **null pointer exception**.

OOP: actual new objects (instances)

```
Student std, std1,  // Declare four
        std2, std3; // vars of type Student.
std = new Student(); // Create a new
                     // object belonging
                     // to the class Student,
                     // store a ref to that
                     // object in the var std.
std1 = new Student(); // Create a
                      //second Student object
std2 = std1;   // Copy the reference value in
               // std1 into the variable std2.
std3 = null;   // Store a null reference in the
               // variable std3.
std.name = "John Smith";   // Set values of some
                           // instance vars
std.ID = Student.getUniqueID();
std1.name = "Mary Jones";
std1.ID = Student.getUniqueID();
     // (Other instance variables have default
     //    initial values of zero.)
```

OOP: objects and pointers

After the computer executes these statements, the situation looks like this:



OOP: objects and assignment

IMPORTANT:

When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.

This is very different from the usual semantics associated with an assignment statement for primitive types:

> **primitive type** values are copied when they are assigned.
>
> **object type** values are NOT copied when they are assigned: only references are changed.

In our example, since `std1.name` was assigned the value "`Mary Jones`", it will also be true that `std2.name` has the value "`Mary Jones`".

In fact, `std1.name` and `std2.name` are just different ways of referring to exactly the same memory location.

Object comparison

When you make a test

```
if (std1 == std2) ...
```

you are testing whether the object references in `std1` and `std2` **point to** exactly the same location in memory;

you are NOT testing whether **the values stored** in the objects are equal.

For example, two distinct `Student` objects with the same name, ID, and test grades would NOT be considered equal by the == operator because distinct objects are stored in distinct memory locations.

Inheritance and polymorphism

A CLASS REPRESENTS A SET OF OBJECTS which share the same structure and behaviors.

The central new idea in object-oriented programming is to allow classes to express the similarities among objects that share some, but not all, of their structure and behavior.

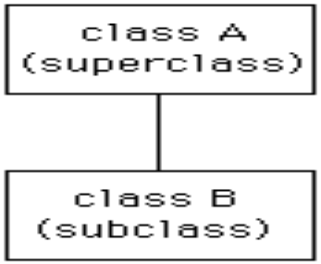
Such similarities can expressed using **inheritance**.

The term "inheritance" refers to the fact that one class can inherit part or all of its structure and behavior from another class.

Inheritance: sub and super

The class that does the inheriting is said to be a **subclass** of the class from which it inherits.

If class B is a subclass of class A, we also say that class A is a **superclass** of class B



Inheritance: sub and super

When you create a new class, you can declare that it is a subclass of an existing class.

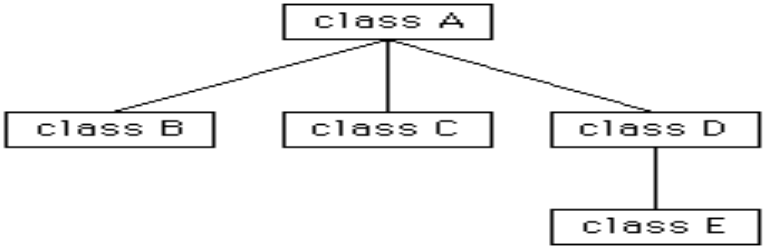
If you are defining a class named "B" and you want it to be a subclass of a class named "A", you would write

```
class B extends A {
  .
  .   // additions to, and
  .   // modifications of,
  .   // stuff inherited from class A
  .
}
```

Several subclasses

Several classes can be declared as subclasses of the same superclass.

The subclasses, which might be referred to as "sibling classes," share some structures and behaviors -- namely, the ones they inherit from their common superclass.

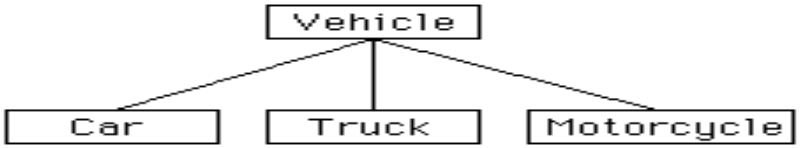


Inheritance can also extend over several "generations" of classes.

In the diagram: class E is a subclass of class D which is itself a subclass of class A.

Example: vehicles

The program could use a class named `Vehicle` to represent all types of vehicles.



Common stuff: the `Vehicle` class could include instance variables such as `registrationNumber` and `owner` and instance methods such as `transferOwnership()`.

Instances: three subclasses of `Vehicle` -- `Car`, `Truck`, and `Motorcycle` -- could then be used to hold variables and methods specific to particular types of vehicles:

the `Car` class might add an instance variable `numberOfDoors`,

the `Truck` class might have `numberOfAxels`,

the `Motorcycle` class could have a boolean variable `hasSidecar`.

Example

```
class Vehicle {
    int registrationNumber;
    Person owner;  // (assuming that a
        //Person class has been defined)
    void transferOwnership(Person newOwner) {
            . . .
    }
. . .
}
class Car extends Vehicle {
    int numberOfDoors;
    . . .
}
class Truck extends Vehicle {
    int numberOfAxels;
    . . .
}
class Motorcycle extends Vehicle {
    boolean hasSidecar;
    . . .
}
```

Vehicle example instances

Suppose that `myCar` is a variable of type `Car` that has been declared and initialized with the statement

```
Car myCar = new Car();
```

Given this declaration, a program could to refer to

```
myCar.numberOfDoors
```

since `numberOfDoors` is an instance variable in the class `Car`.

But since class `Car` extends class `Vehicle`, a car also has all the structure and behavior of a vehicle. This means that

```
myCar.registrationNumber
```

```
myCar.owner
```

```
myCar.transferOwnership()
```

also exist.

Types and subtypes

In the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program.

An object of type `Car` or `Truck` or `Motorcycle` is automatically an object of type `Vehicle`.

This brings us to the following **Important Fact**:

> A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.

In our example: an object of type `Car` can be assigned to a variable of type `Vehicle`. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

Object knows its class

The object "remembers" that it is in fact a `Car`, and not just a `Vehicle`.

Information about the actual class of an object is stored as part of that object.

It is possible to test whether a given object belongs to a given class, using the **`instanceof`** operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by `myVehicle` is in fact a car.

On the other hand, if `myVehicle` is a variable of type `Vehicle` the assignment statement

```
myCar = myVehicle;
```

would be illegal because `myVehicle` could potentially refer to other types of vehicles besides cars.

Similar: the computer will not allow you to assign an `int` value to a variable of type `short`, because not every `int` is a `short`.

Object type casts

If, for some reason, you happen to know that `myVehicle` does in fact refer to a `Car`, you can use the type cast

```
(Car)myVehicle
```

to tell the computer to treat `myVehicle` as if it were actually of type `Car`. So, you could say

```
myCar = (Car)myVehicle;
```

and refer to

```
((Car)myVehicle).numberOfDoors
```

Example: vehicle types and data

```
System.out.println("Vehicle Data:");
System.out.println("Registration number:  " +
                   myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
   System.out.println("Type of vehicle:  Car");
   Car c = (Car)myVehicle;
   System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:” + " Truck");
   Truck t = (Truck)myVehicle;
   System.out.println("Number of axels:  " + t.numberOfAxels);
}
else if (myVehicle instanceof Motorcycle) {
   System.out.println("Type of vehicle: ” + " Motorcycle");
   Motorcycle m = (Motorcycle)myVehicle;
   System.out.println("Has a sidecar:    " + m.hasSidecar);
}
```

Example: shapes

Consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors.

Three classes, `Rectangle`, `Oval`, and `RoundRect`, could be used to represent the three types of shapes.

These three classes could have **a common superclass**, `Shape`, to represent features that all three shapes have in common.

The `Shape` class could include instance variables to represent the color, position, and size of a shape. It could include instance methods for changing the color, position, and size of a shape.

Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color

Shape code

```
class Shape {

 Color color;    // color of shape.  Defined
                 // is defined in package
                 // java.awt.which is assumed
                  // to be imported.

 void setColor(Color newColor) {
    // method to change the color of the shape
    color = newColor; // change value inst var
    redraw();      // redraw shape, in new color
 }

 void redraw() {
       // method for drawing the shape
    ? ? ?  // what commands should go here?
 }

 . . .           // more instance variables and
                 // methods

} // end of class Shape
```

Problem: different ways to draw

The problem is that each different type of shape is drawn differently.

The method `setColor()` can be called for any type of shape. How does the computer know which shape to draw when it executes the `redraw()`?

Informally, we can answer the question like this:

The computer executes `redraw()` by asking the shape to redraw itself.

Every shape object knows what it has to do to redraw itself.

Each class has own redraw

```
class Rectangle extends Shape {
  void redraw() {
  . . .  // commands for drawing a rectangle
  }
  . . . // possibly, more methods and vars
}

class Oval extends Shape {
  void redraw() {
  . . .  // commands for drawing an oval
  }
 . . . // possibly, more methods and vars
}

class RoundRect extends Shape {
  void redraw() {
  . . .  // commands for drawing a rounded
         // rectangle
  }
 . . . // possibly, more methods and vars
}
```

Polymorphism

If **`oneShape`** is a variable of type `Shape`, it could refer to an object of any of the types `Rectangle`, `Oval`, or `RoundRect`.

As a program executes, and the value of **`oneShape`** changes, it could even refer to objects of different types at different times

Whenever the statement

```
oneShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which `oneShape` actually refers.

Suppose the statement is in a loop and gets executed many times. If the value of `oneShape` changes as the loop is executed, it is possible that the very same statement "`oneShape.redraw();`" will call different methods and draw different shapes as it is executed over and over.

We say that the `redraw()` method is **polymorphic**.

A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied.

Polymorphism => Messages

In OOP, calling a method is often referred to as sending a **message** to an object.

The object responds to the message by executing the appropriate method.

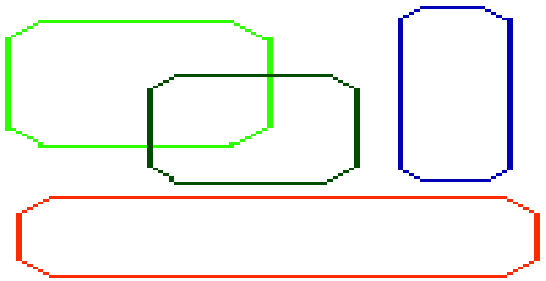
The statement "`oneShape.redraw();`" is a message to the object referred to by `oneShape`. Since that object knows what type of object it is, it knows how it should respond to the message.

From this point of view, the computer always executes "`oneShape.redraw();`" in the same way: by sending a message.

The response to the message depends, naturally, on who receives it. Polymorphism just means that different objects can respond to the same message in different ways.

Polymorphism: adding stuff is easy

If for some reason, I decide that I want to add beveled rectangles to the types of shapes my program can deal with, I can write a new subclass, `BeveledRect`, of class `Shape` and give it is own `redraw()` method.



Automatically, code that I wrote previously -- such as the statement `oneShape.redraw()` -- can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when I wrote the statement!

“this” and “super”

In the statement "`oneShape.redraw();`", the `redraw` message is sent to the object `oneShape`.

However, the **Shape** class contains:

```
void setColor(Color newColor) {
    color = newColor; // change value of
                      // inst var
    redraw(); // redraw shape, which will
              // appear in new color
 }
```

A `redraw` message is sent here, but which object is it sent to?

the `setColor` method is itself a message that was sent to some object. The answer is that the `redraw` message is sent to that same object, the one that received the `setColor` message.

If that object is a rectangle, then it is the `redraw()` method from the `Rectangle` class is executed. If the object is an oval, then it is the `redraw()` method from the `Oval` class.

This and super continued

It also means that the `redraw();` statement in the `setColor()` method does not necessarily call the `redraw()` method in the `Shape` class.

The `redraw()` method that is executed could be in any subclass of `Shape`.

Java has a special name for "**the object that received this message**." Java gives you not one but two names for this object, depending on what you want to do with it.

Whenever an instance method is executed, the system sets up two special variables to refer to the object that received the message. The variables are named **`this`** and **`super`**. You can use these variables in any instance method.

This

The variable "`this`" refers to this object, the one that received this message.

You might need to use `this` if you want to assign the object to a variable or pass it as a parameter.

Consider the `Shape` class, which represents shapes on a computer screen. Suppose that shapes can be "selected," and you want to keep track of which shape is currently selected.

A static variable, `selectedShape`, could be added to the class to keep track of which shape is currently selected. An instance method, `select()`, that is used to select a shape would then assign `this` to `selectedShape`:

```
class Shape {
   static Shape selectedShape = null;    // currently selected
   void select() {  // instance method for selecting this shape
     selectedShape = this;  // record that this is the sel shape
   }
     . . .  // other variables and methods
}
```

This again

Another use of "`this`" is to clear up ambiguities when there are two variables or parameters or local variables of the same name.

For example, consider this slight rewrite of the `setColor()` method from the shape class, in which the parameter `newColor` is renamed to `color`:

```
void setColor(Color color) {
   this.color = color; // change value
                       // of inst variable
   redraw(); // redraw shape, which will appear in new color
}
```

Inside the method setColor(), there are two things with the same name: A parameter and an instance variable are both named "color."

The rule in situations like this is that the parameter hides the instance variable, so that when the name "color" is used by itself, it refers to the parameter.

Fortunately, the instance variable can still be accessed by referring to it with the compound name "**this.color**", which can only mean an instance variable in the object, **this**.

Super

`super` refers to the same object as **`this`**, but it refers to that object treated as if it were a member of the superclass of the class containing the method that you are writing.

`super.x` would refer to an instance variable named `x` in the superclass of the current class.

This can be useful for the following reason: If a class contains an instance variable **with the same name** as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass.

The variable in the subclass does not replace the variable of the same name in the superclass; it merely hides it. The variable from the superclass can still be accessed, using **`super`**.

Super: main use

The major use of **`super`** is to **override a method** with a new method that extends the behavior of the inherited method, instead of replacing that behavior entirely.

Suppose that the class `TextBox` represents a box on the screen where the user can type some input. Let's say that `TextBox` has an instance method called `key` which is called whenever the user presses a key on the keyboard.

The purpose of this method is to add the character that the user has typed to the text box.

Now, suppose I want a subclass, `NumberBox`, of `TextBox` that will let the user type in an integer. I only want to allow digits from 0 to 9 in the box, not letters or punctuation characters.

```
class NumberBox extends TextBox {
   void key(char ch) {
       // user has typed the character ch;
       // enter it only if digit 0-9
      if (ch >= '0' && ch <= '9')
       super.key(ch);
      }
    }
```

...

Abstract classes

`Shape` class with subclasses `Rectangle`, `Oval`, and `RoundRect` representing particular types of shapes.

Each of these classes includes a `redraw()` method. E.g. the `redraw()` method in the `Rectangle` class draws a rectangle.

What does the `redraw()` method in the **`Shape`** class do? How should it be defined?

We should **leave it blank**! The fact is that the class `Shape` represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn.

A `redraw()` method **has to be** in the `Shape` or it would be illegal to call it in the `setColor()` method of the `Shape` class, and it would be illegal to write "`oneShape.redraw();`", where `oneShape` is a variable of type `Shape`.

Nevertheless the version of `redraw()` in the `Shape` class will never be called.

Shape: abstract class

There can never be any reason to construct an actual object of type `Shape`. You can have variables of type `Shape`, but the objects they refer to will always belong to one of the subclasses of `Shape`.

We say that `Shape` is an **abstract class**.

An abstract class is one that is not used to construct objects, but only as a basis for making subclasses.

An abstract class exists only to express the common properties of all its subclasses.

Similarly, we could say that the `redraw()` method in class `Shape` is an **abstract method**, since it is never meant to be called.

Abstract class

You can also tell the computer, syntactically, that they are abstract by adding the modifier "`abstract`" to their definitions.

For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon.

An implementation must be provided for the abstract method in any concrete subclass of the abstract class.

Once you have done this, it becomes illegal to try to create actual objects of type `Shape`, and the computer will report an error if you try to do so.

Abstract Shape: example

```
abstract class Shape {
    Color color;
    void setColor(Color newColor) {
       color = newColor;
       redraw();
    }

    abstract void redraw();

       // abstract method -- must be
       // redefined in concrete
       // subclasses

    . . .
       // more instance variables and
       // methods

} // end of class Shape
```